

(19) 日本国特許庁 (J P)

(12) 公開特許公報 (A)

(11) 特許出願公開番号

特開平11-312152

(43) 公開日 平成11年(1999)11月9日

(51) Int.Cl.<sup>8</sup>

G 0 6 F 15/16

H 0 4 Q 7/38

識別記号

6 2 0

F I

G 0 6 F 15/16

H 0 4 B 7/26

6 2 0 G

1 0 9 M

審査請求 未請求 請求項の数 2 O L (全 28 頁)

(21) 出願番号

特願平10-365666

(22) 出願日

平成10年(1998)12月22日

(31) 優先権主張番号

9 9 5 6 0 6

(32) 優先日

1997年12月22日

(33) 優先権主張国

米国 (U S)

(71) 出願人 590000879

テキサス インストルメンツ インコーポ  
レイテッド

アメリカ合衆国テキサス州ダラス, ノース  
セントラルエクスプレスウェイ 13500

(72) 発明者

マイケル マクマホン

アメリカ合衆国 テキサス州プラノ, メリ  
アン ドライブ 3817

(72) 発明者

マリオン シー. ラインバリー

アメリカ合衆国 テキサス州ダラス, エ  
ヌ. セルバ ドライブ 1250

(74) 代理人

弁理士 浅村 皓 (外 3 名)

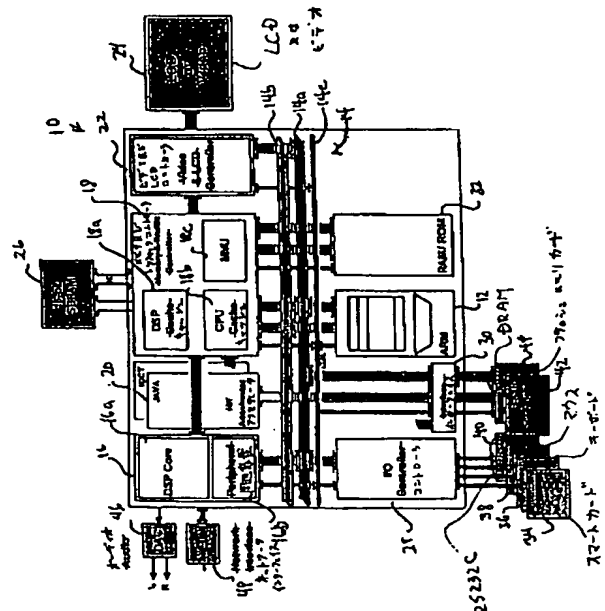
最終頁に続く

(54) 【発明の名称】 移動電子装置および制御方法

(57) 【要約】

【課題】 多数のプロセッサおよびコプロセッサの使用をグレードアップし最適化するデータ処理アーキテクチャを得る。

【解決手段】 複数のプロセッサ12, 16を含むワイヤレスデータプラットフォーム10。タスクが実施される時に情報を通信できるようにプロセッサ間に通信チャネルが開設される。1個のプロセッサで実施されるダイナミッククロスコンパイラ80がコードを別のプロセッサのためのネイティブ処理コードへコンパイルする。ダイナミッククロスリンカー82が他のプロセッサのためにコンパイルされたコードをリンクする。ネイティブコードはそれを包むJ A V A B e a n 9 0 (もしくは他の言語タイプ) を使用してプラットフォームへダウンロードすることもできる。J A V A B e a n は暗号化しセキュリティのためにデジタルサインすることができる。



## 【特許請求の範囲】

【請求項 1】 移動電子装置であって、  
ネイティブコードを実行するコプロセッサと、  
ホストプロセッサシステムおよびプロセッサ独立コード  
に対応するネイティブコードを実行するように作動する  
ホストプロセッサシステムであって、デジタル信号コプ  
ロセッサにより実施されるタスクをダイナミックに変化  
させるように作動するホストプロセッサシステムと、  
前記ホストプロセッサシステムと前記コプロセッサ間の  
通信回路と、  
を含む移動電子装置。

【請求項 2】 移動電子装置の制御方法であって、  
コプロセッサにおいてネイティブコードを実行するステ  
ップと、  
ホストプロセッサシステムにおいてネイティブコードおよ  
びプロセッサ独立コードを実行するステップと、  
デジタル信号コプロセッサにより実施されるタスクを前  
記ホストプロセッサシステムによりダイナミックに変える  
ステップと、  
前記ホストプロセッサシステムと前記コプロセッサ間で  
通信を行うステップと、  
を含む方法。

## 【発明の詳細な説明】

## 【0001】

【発明の属する技術分野】 本発明は一般的に移動電子装  
置に関し、特に移動電子装置用ハードウェアおよびソフ  
トウェアプラットフォームに関する。

## 【0002】

【従来の技術】 装置のパワー、したがって機能、が増強  
されるにつれてハンドヘルド携帯装置の人気の高まって  
きている。近年 PDA (Personal Digital Assistant) が広く使用されており、セル  
ラー電話機と PDA のいくつかの能力を組み合わせたス  
マートホン (Smart phones) が近未来の  
通信に著しいインパクトを与えるものと考えられる。

【0003】 現在ある装置には、音声認識等の、ある個  
別の機能を提供する 1 個以上の DSP (デジタル信号処  
理装置) もしくは他のコプロセッサ、および他のデータ  
処理機能のための汎用プロセッサが内蔵されている。D  
SP 用コードおよび汎用プロセッサ用コードは一般的に  
ROM その他の非揮発性メモリに格納され、それらは容  
易に修正することができない。したがって、改善および  
新しい機能を利用できるようになった時に、しばしば装  
置の能力をグレードアップできないことがある。特に、  
装置内に存在することがある DSP や他のコプロセッサ  
を最大限に使用することができない。

## 【0004】

【発明が解決しようとする課題】 したがって、多数のプ  
ロセッサおよびコプロセッサの使用をグレードアップし  
最適化することができるデータ処理アーキテクチャが

必要とされている。

## 【0005】

【課題を解決するための手段】 本発明において、移動電  
子装置はネイティブコードを実行するコプロセッサと、  
ホストプロセッサシステムに対応するネイティブコード  
およびプロセッサ独立コードを実行するように作動する  
ホストプロセッサシステムとを含んでいる。ホストプロ  
セッサシステムはデジタル信号コプロセッサにより実施  
されるタスクをダイナミックに変えるように作動する。  
通信回路によりホストプロセッサシステムとコプロセッ  
サ間の通信が提供される。

【0006】 本発明により従来技術を凌ぐ著しい利点が  
提供される。それは、ホストプロセッサシステムが、デ  
ジタル信号処理装置であっても良いコプロセッサにより  
実施されるタスクをダイナミックに割り当ててコプロセ  
ッサを完全に使用できるためである。ホストプロセッサ  
システムは、各プロセッサの現在の能力等の、多様な要  
因に応じて複数のコプロセッサの 1 つへ直接ルーチンす  
ることができる。

## 【0007】

【発明の実施の形態】 図 1 に例えばスマートホンや P  
DA を実施するのに使用できる一般的なワイヤレスデー  
タプラットフォームアーキテクチャの好ましい実施例を  
示す。ワイヤレスデータプラットフォーム 10 は、デー  
タバス 14 a、アドレスバス 14 b およびコントロールバ  
ス 14 c を含むバス構造 14 に接続された汎用 (ホス  
ト) プロセッサ 12 を含んでいる。コアプロセッサ 16  
a および周辺インターフェイス 16 b を含む 1 個以上の  
DSP (もしくは他のコプロセッサ) 16 がバス 14 お  
よび DSP キャッシュメモリ 18 a、CPU キャッシュ  
18 b、および MMU (メモリマネジメントユニット)  
18 c を含むメモリおよびトラフィックコントローラ 1  
8 に接続されている。ハードウェアアクセラレータ回路  
20 (JAVA 等のポータブル言語を加速する) および  
ビデオおよび LCD コントローラ 22 もメモリおよびト  
ラフィックコントローラ 18 に接続されている。ビデオ  
および LCD コントローラの出力は LCD もしくはビデ  
オディスプレイ 24 に接続されている。

【0008】 メモリおよびトラフィックコントローラ 1  
8 はバス 14 および、SDRAM (同期ダイナミックラン  
ダムアクセスメモリ) として示す、主記憶装置 26 に  
接続されている。バス 14 は I/O コントローラ 28、  
インターフェイス 30、および RAM/ROM 32 にも  
接続されている。複数の装置をスマートカード 34、キ  
ーボード 36、マウス 38 等のワイヤレスデータプラ  
ットフォーム 10 もしくは USB (ユニバーサルシリアルバ  
ス) ポートもしくは RS232 シリアルポート等の 1 つ  
以上のシリアルポート 40 に接続することができる。イ  
ンターフェイス 30 はフラッシュメモリカード 42 およ  
び/もしくは DRAM カード 44 に接続することができ

る。周辺インターフェイス16bはDSP16をDAC（デジタル／アナログコンバータ）46、ネットワークインターフェイス48もしくは他の装置に接続することができる。

【0009】図1のワイヤレスデータプラットフォーム10は汎用プロセッサ12およびDSP16を利用している。DSP16が特定の固定機能専用とされている現在の装置とは異なり、図1のDSP16は任意数の機能に使用することができる。それにより、ユーザはDSP16の利益を完全に引き出すことができる。

【0010】DSP16を使用できる1つの主要なエリアはマンマシンインターフェイス（MMI）に関連している。重要なのは、音声認識、イメージおよびビデオ圧伸、データ暗号化、テキスト音声変換、等の機能をDSP16を使用してより効率的に実施できることである。本アーキテクチャにより新しい機能や改善をワイヤレスデータプラットフォーム10へ容易に付加することができる。

【0011】ワイヤレスデータプラットフォーム10は一般的なブロック図であり、さまざまに修正することができる。例えば、図1には独立したDSPおよびプロセッサキャッシュ18a、18bが図示されている。当業者にはお判りのように、ユニット型キャッシュも使用することができる。さらに、ハードウェアアクセラレーション（acceleration）回路20はオプションアイテムである。このような装置によりJAVA等の言語の実行が加速されるが、この回路は装置の動作に必要なものではない。さらに、1個のDSPしか図示されていないが、多数のDSP（もしくは他のコプロセッサ）をバスに接続することができる。

【0012】図2にワイヤレスデータプラットフォーム10の機能的ソフトウェアアーキテクチャを示す。このブロック図はJAVAを使用するものと想定しており、JAVA以外の言語も使用することができる。機能的に、ソフトウェアは2つのグループへ分割され、それはホストプロセッサソフトウェアおよびDSPソフトウェアである。ホストソフトウェアは1つ以上のアプレット40を含んでいる。DSP APIクラス42はJAVAアプリケーション用JAVA APIパッケージ、すなわちDSP API50およびホストDSPインターフェイスレイヤ52の機能へアクセスするアプレットである。JAVAバーチャルマシン（VM）44がアプレットを解釈する。JAVAネイティブインターフェイス46は、JAVA VMがホストプロセッサもしくは特定プラットフォームコードを実行する方法である。ネイティブタスク48はJAVAネイティブインターフェイスを使用せずにホストプロセッサ12により実行することができる非JAVAプログラムである。後述するDSP API50は、DSP16の能力を使用するためにコールするホスト12で使用されるAPI（アプリケーション

プログラムインターフェイス）である。ホスト-DSPインターフェイスレイヤ52は他のタスク、あるいはホスト-DSP通信プロトコルを介したチャネルを使用して他のハードウェアと互いに通信するためのAPIをホスト12およびDSP16に提供する。DSPデバイスドライバ54はDSP16と通信するためのホストRTOS56（リアルタイムオペレーティングシステム）用ホストベースデバイスドライバである。ホストRTOS56はアクセラレーテッドテクノロジー社のNUCLEUS PLUS等のオペレーティングシステムである。マイクロソフト社のWINDOWS CE等の非リアルタイムオペレーティングシステムを使用することもできる。DSPライブラリ58にはDSP16で実行するために格納されたプログラムが含まれている。

【0013】DSP側で、DSP16により実行する1つ以上のタスク60をメモリ内に格納することができる。後述するように、DSPの機能がスタティックではなくダイナミックとなるように、タスクを所望によりメモリから出し入れすることができる。DSP側のホスト-DSPインターフェイスレイヤ62はホスト側のホスト-DSPインターフェイスレイヤ52と同じ機能を実施する、すなわちホスト12およびDSP16は通信することができる。DSP RTOS64はDSPプロセッサ用オペレーティングシステムである。ホストデバイスドライバ66はホスト12と通信するDSP RTOS64用DSPベースデバイスドライバである。ホスト-DSPインターフェイス70はDSP16をホスト12に接続する。

【0014】動作において、図2に示すソフトウェアアーキテクチャはDSP16を従来技術のように固定機能装置ではなく、可変機能装置として使用する。したがって、DSP機能は図2のアーキテクチャを内蔵する移動装置へダウンロードして、DSP16がホスト12に対してさまざまな信号処理機能を実施できるようにすることができる。

#### 【0015】DSP-API

DSP-APIはホスト12からDSP16へのデバイス独立インターフェイスを提供する。この機能により、ホスト12にはDSP16にタスクをロードしてスケジュールを行いつこれらのタスクをコントロールして通信する能力が与えられる。API機能には、DSPの利用可能な資源を決定し、ホスト12およびDSPタスクを生成かつコントロールし、ホスト12およびDSPタスク間のデータチャネルを生成かつコントロールし、タスクと通信するための呼出しが含まれる。これらの機能については後述する。各機能はブール結果を返し、それは成功したオペレーションに対するSUCCESS、もしくはFAILUREである。結果がFAILUREであれば、error codeをチェックしてどのエラーが発生したかを確認しなければならない。

【0016】

**DSP\_Get\_MIPS**

【数1】

*BOOL DSP\_Get\_MIPS(T\_DeviceID DevID, U32 \*mips, U16 \*errcode);*

この関数はDSP上で利用できる現在のMIPSを戻す。それはDSP16のMIPS能力マイナスベースMIPS値（付加ダイナミックタスクの無いMIPS値、すなわちカーネルプラスAPIコードプラスドライバ）、マイナスロードされた全てのダイナミックタスクに対するMIPSレーティングの和である。errcodeパラメータは下記の考えられる結果を含む。

【数2】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING

【0017】

【数3】

**DSP\_Get\_Memory\_Available**

*BOOL DSP\_Get\_Memory\_Available(T\_DeviceID DevID, T\_Size \*progmem, T\_Size \*datamem, U16 \*errcode);*

この関数はプログラムメモリおよびデータメモリの両方に対して利用できるメモリ量についてDevIDにより指定されるDSP16に質問する。その結果得られる値はprogmemおよびdatamemパラメータ内に戻される。サイズはT\_DSP\_Words内に指定される。errcodeパラメータは下記の考えられる結果を含む。

【数4】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING

【0018】

【数5】

**DSP\_Alloc\_Mem**

*BOOL DSP\_Alloc\_Mem(T\_DeviceID DevID, U16 mempage, T\_Size size, T\_DSP\_Word \*\*memptr, U16 \*errcode);*

この関数によりDSP16上に1ブロックのメモリが割り当てられる。DevIDはどのデバイス上にメモリを割り当てるかを指定する。mempageはプログラムスペースに対しては0であり、データスペースに対しては1である。sizeパラメータはT\_DSP\_Words内のメモリブロックサイズを指定する。戻されるmemptrはDSP16上のメモリブロックへのポインター、もしくは失敗時のNULLである。errcodeパラメータは下記の考えられる結果を含む。

【数6】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_INVALID\_MEMPAGE  
DSP\_NOT\_ENOUGH\_MEMORY

【0019】

【数7】

**DSP\_Free\_Mem**

*BOOL DSP\_Free\_Mem(T\_DeviceID DevID, U16 mempage, T\_DSP\_Word \*memptr, U16 \*errcode);*

この関数はDSP Alloc\_Mem関数を割り当てられたDSP上の1ブロックのメモリを解放する。DevIDはどのデバイス上にメモリが常駐するかを指定する。mempageはプログラムスペースに対しては0であり、データスペースに対しては1である。memptrパラメータはメモリブロックへのポインターである。errcodeパラメータは下記の考えられる結果を含む。

【数8】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_INVALID\_MEMPAGE  
DSP\_MEMBLOCK\_NOT\_FOUND

【0020】

【数9】

**DSP\_Get\_Code\_Info**

*BOOL DSP\_Get\_Code\_Info(char \*Name, T\_CodeHdr \*codehdr, U16 \*errcode);*

この関数はDSPライブラリテーブルへアクセスし、Nameパラメータにより指定されるDSP関数コードのコードヘッダーを戻す。戻されると、codehdrパラメータにより指示される位置はコードヘッダー情報を含む。errcodeパラメータは下記の考えられる結果を含む。

#### DSP\_Link\_Code

```
BOOL DSP_Link_Code(T_DeviceID DevID, T_CodeHdr *codehdr,
T_TaskCreate *tcs, U16 *errcode);
```

この関数はDevIDにより指定されるDSP上の指示されたアドレスで走るようにDSP関数コードをリンクする。codehdrパラメータは関数のコードヘッダーを指示する。ダイナミッククロスリンカーはコードヘッダー内およびコード(COFFファイル)内の情報に基づいてコードをリンクする。ダイナミッククロスリンカーは必要に応じてメモリを割り当て、DSP16にコードをリンクしロードする。tcsパラメータはDSP\_Create\_Task関数において必要なタスク生成構造へのポインタである。DSP\_Link\_Codeはタスクの生成に備えて構造のコードエントリポインタ、優先順位、およびクオンタム(quantu

#### DSP\_Put\_BLOB

```
BOOL DSP_Put_BLOB(T_DeviceID DevID, T_HostPtr srcaddr, T_DSP_Ptr
destaddr, U16 mempage, T_Size size, U16 *errcode);
```

この関数は指定されたBLOB(Binary Large Object)をDSP16へコピーする。DevIDはどのDSP16へオブジェクトをコピーするかを指定する。srcaddrパラメータはホストメモリ内のオブジェクトへのポインタである。destaddrはDSP16上のオブジェクトをコピーする位置へのポインタである。mempageはプログラムスペースに対しては0であり、データスペースに対しては1である。サイズパラメータはTDSP\_Words内の

#### DSP\_Create\_Task

```
BOOL DSP_Create_Task(T_DeviceID DevID, T_TaskCreate *tcs, T_TaskID
*TaskID, U16 *errcode);
```

#### DSP

Create\_Taskは、タスクパラメータおよびDSPのプログラムスペース内のコード位置が与えられたらタスクを生成するようDSP16に要求する。タスク

【数10】

```
DSP_SUCCESS
DSP_NAMED_FUNC_NOT_FOUND
```

【0021】

【数11】

m) フィールドを充填する。errcodeパラメータは下記の考えられる結果を含む。

【数12】

```
DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_NOT_ENOUGH_PROG_MEMORY
DSP_NOT_ENOUGH_DATA_MEMORY
DSP_COULD_NOT_LOAD_CODE
```

【0022】

【数13】

オブジェクトのサイズを指定する。errcodeパラメータは下記の考えられる結果を含む。

【数14】

```
DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_INVALID_MEMPAGE
```

【0023】

【数15】

生成構造を表1に示す。

【0024】

【表1】

表 1 タスク生成構造

データタイプ	フィールド名	説明
T_DSP_Name	Name	User defined name for the task. (タスクのユーザ定義名)
U32	MIPS	MIPS used by the task. (タスクが使用するMIPS)
T_ChanelID	ChanIn	The channel ID used for task input. (タスク入力に使用するチャンネルID)
T_ChanelID	ChanOut	The channel ID used for task output. (タスク出力に使用するチャンネルID)
T_StreamID	StrmIn	The stream ID used for task input. (タスク入力に使用するストリームID)
T_StreamID	StrmOut	The stream ID used for task output. (タスク出力に使用するストリームID)
U16	Priority	The task's priority. (タスクの優先順位)
U32	Quantum	The task's timeslice in system ticks. (システムチック内のタスクのタイムスライス)
T_Size	StackReq	The amount of stack required. (所有タスク量)
T_DSP_Ptr	MsgHandler	Pointer to code to handle messages to the task. (タスクへのメッセージを処理するコードへのポインタ)
T_HOST_Ptr	CallBack	Pointer to Host code to handle messages from the task. (タスクからのメッセージを処理するホストコードへのポインタ)
T_DSP_Ptr	Create	Pointer to code to execute when task is created. (タスク生成時に実行するコードへのポインタ)
T_DSP_Ptr	Start	Pointer to code to execute when task is started. (タスク開始時に実行するコードへのポインタ)
T_DSP_Ptr	Suspend	Pointer to code to execute when task is suspended. (タスク保留時に実行するコードへのポインタ)
T_DSP_Ptr	Resume	Pointer to code to execute when task is resumed. (タスク再開時に実行するコードへのポインタ)
T_DSP_Ptr	Stop	Pointer to code to execute when task is stopped. (タスク停止時に実行するコードへのポインタ)

【0025】タスクが生成されると、Create エントリーポイントが呼び出され、任意の必要な予備初期化を行う機会をタスクに与える。Create, Suspend, Resume, および Stop エントリーポイントは NULL とすることができる。その結果得られる TaskID はデバイス ID (DevID) と DSP のタスク ID の両方を含んでいる。TaskID が NULL であれば、生成は失敗である。errcode パラメータは下記の考えられる結果を含む。

#### DSP\_Start\_Task

*BOOL DSP\_Start\_Task(T\_TaskID TaskID, U16 \*errcode);*

この関数は TaskID により指定される DSP タスクを開始する。タスクの Start エントリーポイントで実行が開始する。errcode パラメータは下記の考えられる結果を含む。

【数 18】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_TASK\_NOT\_FOUND

【数 16】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_INVALID\_PRIORITY  
DSP\_CHANNEL\_NOT\_FOUND  
DSP\_ALLOCATION\_ERROR

【0026】

【数 17】

【0027】

【数 19】

**DSP\_Suspend\_Task**

*BOOL DSP\_Suspend\_Task(T\_TaskID TaskID, U16 \*errcode);*

この関数はTaskIDにより指定されたDSPタスクを保留する。保留される前に、任意の必要なハウスキューピングを実施する機会をタスクに与えるためにタスクのSuspendエントリーポイントが呼び出される。errcodeパラメータは下記の考えられる結果を含む。

【数20】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_TASK\_NOT\_FOUND

【0028】

【数21】

**DSP\_Resume\_Task**

*BOOL DSP\_Resume\_Task(T\_TaskID TaskID, U16 \*errcode);*

この関数はDSP Suspend\_Taskにより保留されたDSPタスクを再開する。再開される前に、任意の必要なハウスキューピングを実施する機会をタスクに与えるためにタスクのResumeエントリーポイントが呼び出される。errcodeパラメータは下記の考えられる結果を含む。

【数22】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_TASK\_NOT\_FOUND  
DSP\_TASK\_NOT\_SUSPENDED

【0029】

【数23】

**DSP\_Delete\_Task**

*BOOL DSP\_Delete\_Task(T\_TaskID TaskID, U16 \*errcode);*

この関数はTaskIDにより指定されたDSPタスクを削除する。削除する前に、任意の必要なクリーンアップを実施する機会をタスクに与えるためにタスクのStopエントリーポイントが呼び出される。それはタスクにより割り当てられた任意のメモリを解放し、タスクが取得した任意の資源を戻すことを含まなければならない。errcodeパラメータは下記の考えられる結果を含む。

【数24】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_TASK\_NOT\_FOUND

【0030】

【数25】

**DSP\_Change\_Task\_Priority**

*BOOL DSP\_Change\_Task\_Priority(T\_TaskID TaskID, U16 newpriority, U16 \*oldpriority, U16 \*errcode);*

この関数はTaskIDにより指定されたDSPタスクの優先順位を変える。優先順位はnewpriorityへ変えられる。newpriorityの考えられる値はRTOS依存である。戻ると、oldpriorityパラメータはタスクの前の優先順位に設定される。errcodeパラメータは下記の考えられる結果を含む。

【数26】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_TASK\_NOT\_FOUND  
DSP\_INVALID\_PRIORITY

【0031】

【数27】

**DSP\_Get\_Task\_Status**

*BOOL DSP\_Get\_Task\_Status(T\_TaskID TaskID, U16 \*status, U16 \*priority, T\_ChainID \*Input, T\_ChainID \*Output, U16 \*errcode);*

この関数はTaskIDにより指定されたDSPタスクの状態を戻す。statusは下記の値の1つを含む。

【数28】

DSP\_TASK\_RUNNING  
 DSP\_TASK\_SUSPENDED  
 DSP\_TASK\_WAITFOR\_SEM  
 DSP\_TASK\_WAITFOR\_QUEUE  
 DSP\_TASK\_WAITFOR\_MSG

む。

【数 29】

DSP\_SUCCESS  
 DSP\_DEVID\_NOT\_FOUND  
 DSP\_DEVID\_NOT\_RESPONDING  
 DSP\_TASK\_NOT\_FOUND

【0032】 *priority* パラメータはタスクの優先順位を含み、*Input* および *Output* パラメータはタスクの、それぞれ、入力および出力 ID を含む。*errcode* パラメータは下記の考えられる結果を含む。

【0033】

【数 30】

**DSP\_Get\_ID\_From\_Name**

*BOOL DSP\_Get\_ID\_From\_Name(T\_DeviceID DevID, T\_DSP\_Name Name, T\_DSP\_ID \*ID, U16 \*errcode);*

この関数は DSP16 上の名前を付けたオブジェクトの ID を戻す。名前を付けたオブジェクトはチャンネル、タスク、メモリブロック、もしくは任意他のサポートされた名前を付けた DSP オブジェクトとすることができ、*errcode* パラメータは下記の考えられる結果を含む。

DSP\_SUCCESS  
 DSP\_DEVID\_NOT\_FOUND  
 DSP\_DEVID\_NOT\_RESPONDING  
 DSP\_NAME\_NOT\_FOUND

【0034】

【数 32】

【数 31】

**DSP\_Dbg\_Read\_Mem**

*BOOL DSP\_Dbg\_Read\_Mem(DEVICE\_ID DevID, U8 mempage, DSP\_PTR addr, U32 count, DSP\_WORD \*buf, U16 \*errcode);*

この関数はメモリの 1 ブロックを要求する。*mempage* はプログラムメモリ (0) もしくはデータメモリ (1) を指定する。*addr* パラメータはメモリ開始アドレスを指定し、*count* はどれだけ多くの *T\_DSP\_Words* を読み出すかを示す。*buf* パラメータはメモリをコピーしなければならないバッファを提供した呼び出し者へのポインターである。*errcode* パラメータは下記の考えられる結果を含む。

【数 33】

DSP\_SUCCESS  
 DSP\_DEVID\_NOT\_FOUND  
 DSP\_DEVID\_NOT\_RESPONDING  
 DSP\_INVALID\_MEMPAGE

【0035】

【数 34】

**DSP\_Dbg\_Write\_Mem**

*BOOL DSP\_Dbg\_Write\_Mem(T\_DeviceID DevID, U16 mempage, T\_DSP\_Ptr addr, T\_Count count, T\_DSP\_Word \*buf, U16 \*errcode);*

この関数はメモリの 1 ブロックを書き込む。*mempage* はプログラムメモリ (0) もしくはデータメモリ (1) を指定する。*addr* パラメータはメモリ開始アドレスを指定し、*count* はどれだけ多くの *T\_DSP\_Words* を書き込むかを示す。*buf* パラメータは書き込むメモリを含むバッファへのポインターである。*errcode* パラメータは下記の考えられる結果を含む。

【数 35】

DSP\_SUCCESS  
 DSP\_DEVID\_NOT\_FOUND  
 DSP\_DEVID\_NOT\_RESPONDING  
 DSP\_INVALID\_MEMPAGE

【0036】

【数 36】

**DSP\_Dbg\_Read\_Reg**

*BOOL DSP\_Dbg\_Read\_Reg(T\_DeviceID DevID, U16 RegID, T\_DSP\_Word \*regvalue, U16 \*errcode);*

この関数は DSP レジスタを読み出し *regvalue* 内の値を戻す。*RegID* パラメータはどのレジスタを

戻すべきかを指定する。*RegID* が -1 であれば、全てのレジスタ値が戻される。バッファを提供した発呼



者へのポインターである `regvalue` パラメータは、全ての値を保持するのに十分な記憶装置を指示しなければならない。レジスタ ID は特定 DSP 向けであり特定のインプリメンテーションに依存する。 `errcode` パラメータは下記の考えられる結果を含む。

【数 37】

#### DSP\_Dbg\_Write\_Reg

*BOOL DSP\_Dbg\_Write\_Reg(T\_DeviceID DevID, U16 RegID, T\_DSP\_Word regvalue, U16 \*errcode);*

この関数は DSP レジスタに書き込む。 `RegID` パラメータはどのレジスタを修正するかを指定する。 `regvalue` は書き込む新しい値を含んでいる。レジスタ ID は特定 DSP 向けであり特定のインプリメンテーションに依存する。 `errcode` パラメータは下記の考えられる結果を含む。

【数 39】

#### DSP\_Dbg\_Set\_Break

*BOOL DSP\_Dbg\_Set\_Break(T\_DeviceID DevID, DSP\_Ptr addr, U16 \*errcode);*

この関数は所与のコードアドレス (`addr`) におけるブレイクポイント (`break point`) を設定する。 `errcode` パラメータは下記の考えられる結果を含む。

【数 41】

#### DSP\_Dbg\_Clr\_Break

*BOOL DSP\_Dbg\_Clr\_Break(T\_DeviceID DevID, T\_DSP\_Ptr addr, U16 \*errcode);*

この関数は所与のコードアドレス (`addr`) において DSP

`Dbg_Set_Break` により予め設定されたブレイクポイントをクリアする。 `errcode` パラメータは下記の考えられる結果を含む。

【数 43】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_BP\_DID\_NOT\_EXIST

#### 【0040】 DSP デバイスドライバ

DSP デバイスドライバ 54 はホスト 12 から DSP 16 への通信を処理する。ドライバ関数はホスト-DSP 通信プロトコルに明記された通信要求を取り上げ、利用可能なハードウェアインターフェイスを介した情報の伝送を処理する。デバイスドライバは RTOS 依存かつ通信ハードウェア依存である。

#### 【0041】 DSP ライブラリ

DSP ライブラリ 58 は DSP 16 へダウンロードして実行することができるコードのブロックを含んでいる。

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_INVALID\_REGISTER

【0037】

【数 38】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING  
DSP\_INVALID\_REGISTER

【0038】

【数 40】

DSP\_SUCCESS  
DSP\_DEVID\_NOT\_FOUND  
DSP\_DEVID\_NOT\_RESPONDING

【0039】

【数 42】

ダイナミッククロスリンカーが全てのアドレス参照 (`reference`) を決定できるように、コードの各ブロックは予め非リンク、すなわちライブラリとして再配置可能にリンクされている。各コードブロックは DSP MIPS (百万命令/秒)、優先順位、タイムスライスオンタム、およびメモリに対するブロックの要求に関する情報も含んでいる。コードブロックヘッダーのフォーマットを表 2 に示す。プログラムメモリおよびデータメモリサイズは、DSP がタスクのメモリ要求をサポートできるかどうかの迅速なチェックをホスト 12 に与えるための近似値である。十分なスペースがあるようであれば、ダイナミッククロスリンカーはコードのリンクおよびロードを試みることができる。ページアライメント (`alignment`) および連続性 (`contiguity`) の要求により、ダイナミッククロスリンカーはそれでも目的を果たせないことがある。好ましい実施例では、コードはバージョン 2 COFF ファイルフォーマットである。

【0042】

【表 2】

表2 コードブロックヘッダー

データタイプ	フィールド名	説明
U16	Processor	The target processor type. (ターゲットプロセッサタイプ)
T_DSP_Ptr	Name	Task's name. (タスク名)
U32	MIPS	Worst case MIPS required by the task. (タスクが要求する最悪時のMIPS)
T_Size	ProgSize	Total program memory size needed. (所用総プログラムメモリサイズ)
T_Size	DataSize	Total data memory size needed. (所用総データメモリサイズ)
T_Size	InFrameSize	Size of a frame in the task's input channel. (タスク入力チャネル内のフレームサイズ)
T_Size	OutFrameSize	Size of a frame in the task's output channel. (タスク出力チャネル内のフレームサイズ)
T_Size	InStrmSize	Size of the task's input stream FIFO. (タスクの入力ストリームFIFOのサイズ)
T_Size	OutStrmSize	Size of the task's output stream FIFO. (タスクの出力ストリームFIFOのサイズ)
U16	Priority	Task's priority. (タスクの優先順位)
U32	Quantum	Task's time slice quantum (number of system ticks). (タスクのタイムスライスクォンタム)(システムチック数)
T_Size	StackReq	Stack required. (所用スタック)
T_Size	CoffSize	Total size of the COFF file. (COFFファイルの総サイズ)
T_DSP_Ptr	MsgHandler	Offset to a message handler entry point for the task. (タスクのメッセージハンドラーエントリポイントへオフセット)
T_DSP_Ptr	Create	Offset to a create entry point that is called when the task is created. (タスク生成時に呼び出される生成エントリポイントへオフセット)
T_DSP_Ptr	Start	Offset to the start of the task's code. (タスクコードの開始へオフセット)
T_DSP_Ptr	Suspend	Offset to a suspend entry point that is called prior to the task being suspended. (保留されるタスクよりも前に呼び出される保留エントリポイントへオフセット)
T_DSP_Ptr	Resume	Offset to a resume entry point that is called prior to the task being resumed. (再開されるタスクよりも前に呼び出される再開エントリポイントへオフセット)
T_DSP_Ptr	Stop	Offset to a stop entry point that is called prior to the task being deleted. (削除されるタスクよりも前に呼び出される停止エントリポイントへオフセット)
T_Host_Ptr	CoffPtr	Pointer to the location of the COFF data in the DSP Library. (DSPライブラリ内のCOFFデータ位置へのポインタ)

【0043】ポータブルコードのリンクされたターゲットコードへの変換

JAVAコード等のポータブル(プロセッサ独立)コードをリンクされたターゲットコードへ変換する手順を図3に示す。この手順は2つの機能、ダイナミッククロスコンパイラー80およびダイナミッククロスリンカー82、を使用する。各機能はホストプロセッサ12上で実現される。好ましい実施例では、ダイナミッククロスリンカーはDSP-AP1の一部である。クロスコンパイラーはDSP-AP1の一部とすることもできる。

【0044】ダイナミッククロスコンパイラー80はポータブルコードをリンクされない、実行可能なターゲットプロセッサコードへ変換する。ダイナミッククロスリンカー82は非リンク、実行可能なターゲットプロセッサコードをリンクされた、実行可能なターゲットプロセッサコードへ変換する。そうするために、DSP16上

ヘローディングする前に、1ブロックのコード内のアドレスを決定しなければならない。ダイナミッククロスリンカー82は関数のコードセグメントおよびデータセグメントをリンクし、DSP16上のメモリを割り当て、コードおよび一定のデータをDSP16へロードする。コードを実行するターゲットプロセッサ(すなわち、DSP16)とは異なるプロセッサ(すなわち、ホストプロセッサ12)で関数(コンパILINGおよびリンキング)が生じるため、この関数は“クロス”コンパILINGおよび“クロス”リンキングと言われる。

【0045】ダイナミッククロスコンパイラー80はユーザもしくはユーザエージェント(ブラウザ等)がオンデマンドでロードした予めリンクされていないコードを受け取る。コードは(1)コードの“タグ”部を識別するか、もしくは(2)DSP16で実行する適性について非タグコードセグメントを解析するために処理され

る。ソースコードのタグ部はその中に埋め込まれた“<start DSP code>”および“<end DSP code>”等の所定のマーカによりDSPへターゲット可能なソースを描くことができる。タグ部が直接もしくは解析により識別される場合には、DSP16の現在の処理状態に基づいてクロスコンパイルするかどうか判断される。コンパイルすると判断されれば、コードのその部分は既知のコンパILING方法を使用して非リンク、実行可能ターゲットプロセッサコードを出力するソフトウェアをコンパILINGすることにより処理される。コンパイルしないという判断は、例えば、DSP16により他のタスクが実行されているため、DSPが利用できる容量（一般的に、利用可能MIPS一百万命令/秒といわれる）が不十分であるか、利用可能なメモリが不十分である場合になされる。コンパイルされたコードはダイナミッククロスリンカー82へ通してDSP16において即座に使用するか、あるいはDSPライブラリ58内に保存することができる。

【0046】ダイナミッククロスリンカー82は予め非リンクコードを受け取り、それは（1）ホストプロセッサ12と関連してスタティックに格納されるか、（2）ネットワークコネクション（インターネット等のグローバルネットワークを含む）を介してホストプロセッサ12へダイナミックにダウンロードされるか、あるいは（3）ダイナミッククロスコンパイラ80によりダイナミックに発生される。ダイナミッククロスリンカー82はランタイムに決定されるDSP16のメモリ開始アドレスの入力コードをリンクする。メモリ開始アドレスはホストプロセッサ12もしくはDSP16により格納され管理されるメモリマップもしくはメモリテーブルから決定することができる。ダイナミッククロスリンカー82はコード内の基準メモリ位置をDSP内の実際のメモリ位置へ変換する。これらのメモリ位置は、例えば、コード内の分岐アドレスもしくはコード内のデータ位置参照（reference）を含むことができる。

【0047】実施例では、ポータブルコードはリンクされているかどうかを含めたコードに関する全情報を含むCOFF（共通オブジェクトファイルフォーマット）内にある。リンクされていないければ、コードをリンクするのに変えなければならないアドレスをシンボルテーブルが定義する。

【0048】前記した変換プロセスには従来技術に較べていくつかの著しい利点がある。第1に、ダイナミッククロスコンパイラ80はダウンロードされたポータブルコードをどこで実行するかについてランタイム判断を行うことができる。例えば、多数のターゲットプロセッサ（2台のDSP16等）を有するシステムでは、ダイナミッククロスコンパイラ80は利用可能な資源もしくは能力に基づいてポータブルコードを任意の1台のターゲットプロセッサへコンパイルすることができる。ダ

イナミッククロスリンカー82はリロケータブルコードをサポートしないターゲットプロセッサでランするようコードをリンクする。コードはランタイムにリンクされるため、DSP16（もしくは、他のターゲットプロセッサ）内のメモリ位置は保存する必要がなく、デバイス内の全ての計算資源を最適効率で使うことができる。コンパILINGはプラットフォーム10のアーキテクチャの知識により達成されるため、一方もしくは両方のプロセッサのインテグリエントキャッシュアーキテクチャ等の特定プロセッサおよびプラットフォーム向けの特徴をコンパILINGに利用することができる。

【0049】したがって、DSP16はその処理能力を完全に使用するようにダイナミックに変えられるさまざまな機能を有することができる。例えば、ユーザは音声認識を含むユーザインターフェイスをロードしたいことがある。その時、ホストプロセッサはソフトウェアをダウンロードしてDSP16で実行する音声認識ソフトウェアをダイナミックにクロスコンパイルおよびクロスリンクする。あるいは、DSP16の現在状態に基づいてDSPライブラリ58内の予めコンパイルされたソフトウェアをダイナミックにクロスリンクして実行することができる。

【0050】ホストデバイスドライバ  
ホストデバイスドライバはDSP16からホスト12への通信を処理する。ドライバ機能はホスト-DSP通信プロトコルに明記された通信要求を取り入れて利用可能なハードウェアインターフェイスを介した情報の伝送を処理する。デバイスドライバはRTOS依存かつ通信ハードウェア依存である。

【0051】ホスト-DSP通信プロトコル（ホスト-DSPインターフェイスレイヤ）  
ホスト-DSP通信プロトコルはホスト12とDSP16間のコマンドおよびデータの通信を支配する。通信はいくつかのパス、メッセージ、データチャネル、およびストリーム、からなる。メッセージは初期化パラメータおよびコマンドをタスクへ送るのに使用される。データチャネルはタスク間およびDSP16とホスト12間で大量のデータをデータフレームの形で運ぶ。ストリームはタスク間およびDSP16とホスト12間でストリームとされたデータを通すのに使用される。

【0052】メッセージ  
各タスクはメッセージを処理するメッセージハンドラーへのエン트리ポイントを有する。メッセージはユーザ定義であり、タスク機能のための初期化パラメータおよびタスクをコントロールするコマンドを含む。タスクはその生成時に指定されるコールバックを介してホスト12へメッセージを送る。タスクメッセージハンドラーのプロトタイプおよびホストコールバックのプロトタイプをここに示す。

【数44】

```

void TaskMsgHandler(T_ReplyRef replyref, T_MsgID MsgID, T_Count count,
    T_DSP_Word *buf);
void HostCallBack(T_ReplyRef replyref, T_MsgID MsgID, T_Count count,
    T_DSP_Word *buf);

```

【0053】replyrefパラメータは送信者への返答を返送するのに使用されるインブルメンテーション依存基準値である。各Send\_Messageコールについて、受信者はreplyrefパラメータを使用

してReply\_Messageを呼び出さなければならない。実際のメッセージは次のようである。

【数45】

Sent message:	MsgPktFlag	taskid	replyref	msgid	count	buf[.....]
Reply message:	MsgPktFlag	-1	replyref	msgid	count	buf[.....]

マルチワードデータは最下位語が最初に送られる。

【0054】Send\_Message関数内の0のTaskIDはシステムレベルメッセージを示す。システムレベルメッセージはDSP-API関数を実施するの

**Send\_Message**

```

BOOL Send_Message(T_TaskID TaskID, T_MsgID MsgID, T_Count count,
    T_DSP_Word *msgbuf, T_DSP_Word *replybuf, T_Size replybufsize,
    T_Count replycount, U16 *errcode);

```

この関数はTaskIDにより指定されるタスクヘユーザ定義メッセージを送る。MsgIDはメッセージを定義し、msgbufは実際のメッセージデータを含んでいる。メッセージサイズはcountT\_DSP\_Wordsである。メッセージへの返答はreplybufパラメータ内に含まれ、それは呼び出し者により提供されるサイズreplybufsizeのバッファを指示する。それは特定のメッセージに対する返答を処理するのに十分なサイズでなければならない。errcode

**Reply\_Message**

```

BOOL Reply_Message(T_ReplyRef replyref, T_Count count, T_DSP_Word *buf,
    U16 *errcode);

```

この関数はメッセージへ返答するのに使用される。replyrefはオリジナルメッセージの送信者へ返答を返送するのに使用される参照(reference)であり、特定インブルメンテーション向けである。返答はbufパラメータ内に含まれそのサイズはT\_DSP\_Wordsである。errcodeパラメータは下記の考えられる結果を含む。

【数49】

```

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_BAD_REPLY_REF

```

【0057】チャンネル

チャンネルの概念は1つのプロセッサから別のプロセッサへ、あるいは同じプロセッサ上のタスク間でフレームベースデータを送信するのに使用される。生成されると、

に使用される。

【0055】メッセージ関数を下記に示す。

【数46】

eパラメータは下記の考えられる結果を含む。

【数47】

```

DSP_SUCCESS
DSP_DEVID_NOT_FOUND
DSP_DEVID_NOT_RESPONDING
DSP_TASK_NOT_FOUND

```

【0056】

【数48】

チャンネルはデータを含むように指定された数およびサイズのフレームを割り当てる。最初に、チャンネルは空フレームのリストを含んでいる。データを生じるタスクはデータを書き込む空フレームを要求し、書き込まれるとフレームはチャンネルへ戻される。データを消費するタスクはチャンネルから完全なフレームを要求し、空になるとフレームはチャンネルへ戻される。フレームバッファのこの要求および戻しにより、最小限のコピーでデータを動かすことができる。

【0058】各タスクは指定された入力および出力チャンネルを有する。チャンネルが生成されると、それは1つのタスクへの入力、およびもう1つのタスクへの出力として指示される。チャンネルのIDにはデバイスIDが含まれ、チャンネルはプロセッサ間でデータを通すことができる。ホスト-DSPインターフェイスを横切るチャンネルデータフローは下記のようなものである。

【数50】

ChanPktFlag	Channel ID	Count	Data[...]
-------------	------------	-------	-----------

チャンネル関数を下記に示す。

【数51】

【0059】

**Create\_Channel**

**BOOL Create\_Channel(T\_DeviceID DevID, T\_Size framesize, T\_Count numframes, T\_ChannelID \*ChannelID, U16 \*errcode);**

この関数はデータフレームベース通信チャンネルを生成する。それはカウントおよびサイズが、それぞれ、numframesおよびframesize内に指定されている1組のフレームバッファのコントロールを維持するチャンネルコントロール構造を生成する。生成されると、チャンネルはデータフレームを割り当てそれらをその空フレームリストへ加える。ChannelIDは新しいチャンネルのIDを返す。DevIDが呼出プロセッサのものでなければ、呼出プロセッサとDevIDプロセッサの両方でチャンネルコントロール構造が生成され、通信インターフェイスを横切るデータフローがコントロー

ルされる。errcodeパラメータは下記の考えられる結果を含む。

【数52】

CHAN\_SUCCESS  
CHAN\_DEVID\_NOT\_FOUND  
CHAN\_DEVID\_NOT\_RESPONDING  
CHAN\_ALLOCATION\_ERROR

【0060】

【数53】

**Delete\_Channel**

**BOOL Delete\_Channel(T\_ChannelID ChannelID, U16 \*errcode);**

この関数はChannelIDにより指定される既存のチャンネルを削除する。errcodeパラメータは下記の考えられる結果を含む。

【0061】

【数55】

【数54】

CHAN\_SUCCESS  
CHAN\_DEVID\_NOT\_FOUND  
CHAN\_DEVID\_NOT\_RESPONDING  
CHAN\_CHANNEL\_NOT\_FOUND

**Request\_Empty\_Frame**

**BOOL Request\_Empty\_Frame(T\_LocalChanID Chn, T\_DSP\_Word \*\*bufptr, BOOL WaitFlag, U16 \*errcode);**

この関数は指定されたローカルチャンネルIDから空フレームを要求する。ChnがNULLであれば、タスクの出力チャンネルが使用される。戻る時に、bufptrはフレームバッファへのポインタを含んでいる。WaitFlagがTRUEであり、かつ利用可能なフレームバッファがなければ、呼出者はバッファを利用できるようにまで保留される。WaitFlagがFALSEであれば、機能はとにかく戻る。errcode

eパラメータは下記の考えられる結果を含む。

【数56】

CHAN\_SUCCESS  
CHAN\_CHANNEL\_NOT\_FOUND  
CHAN\_BUFFER\_UNAVAILABLE

【0062】

【数57】

**Return\_Full\_Frame**

**BOOL Return\_Full\_Frame(T\_LocalChanID Chn, T\_DSP\_Word \*bufptr, U16 \*errcode);**

一度タスクがフレームバッファを満たすと、この関数を使ってチャンネルへ戻される。bufptrにより指示されるバッファは指定されたチャンネルIDへ戻され

る。ChnがNULLであれば、タスクの出力チャンネルが使用される。errcodeパラメータは下記の考えられる結果を含む。

【数 5 8】

CHAN\_SUCCESS  
CHAN\_CHANNEL\_NOT\_FOUND  
CHAN\_BUFFER\_CTRL\_ERROR

**Request\_Full\_Frame**

*BOOL Request\_Full\_Frame(T\_LocalChanID Chn, T\_DSP\_Word \*\*bufptr,  
BOOL WaitFlag, U16 \*errcode);*

この関数は指定されたローカンチャンネルIDからデータの全フレームを要求する。ChnがNULLであれば、タスクの入力チャンネルが使用される。戻る時に、bufptrパラメータはフレームバッファへのポインタを含んでいる。WaitFlagがTRUEであり、かつ利用可能な完全なフレームバッファがなければ、呼び出し者はバッファが利用できるようになるまで保留される。WaitFlagがFALSEであれば、関数

**Return\_Empty\_Frame**

*BOOL Return\_Empty\_Frame(T\_LocalChanID Chn, T\_DSP\_Word \*bufptr, U16  
\*errcode);*

タスクはフレームバッファからのデータを使用したら、この関数を使用してバッファをチャンネルへ戻さなければならない。bufptrにより指示されるバッファは指定されたチャンネルIDへ戻される。ChnがNULLであれば、タスクの入力チャンネルが使用される。errcodeパラメータは下記の考えられる結果を含む。

**Set\_Task\_Input\_Channel**

*BOOL Set\_Task\_Input\_Channel(T\_Task \*TaskID, T\_ChannelID ChanID, U16  
\*errcode);*

この関数はタスクの入力チャンネルを指定されたチャンネルIDに設定する。errcodeパラメータは下記の考えられる結果を含む。

【数 6 4】

CHAN\_SUCCESS  
CHAN\_DEVID\_NOT\_FOUND  
CHAN\_DEVID\_NOT\_RESPONDING  
CHAN\_TASK\_NOT\_FOUND  
CHAN\_CHANNEL\_NOT\_FOUND

**Set\_Task\_Output\_Channel**

*BOOL Set\_Task\_Output\_Channel(T\_Task \*TaskID, T\_ChannelID ChanID, U16  
\*errcode);*

この関数はタスクの出力チャンネルを指定されたチャンネルIDに設定する。errcodeパラメータは下記の考えられる結果を含む。

【数 6 6】

【0 0 6 3】

【数 5 9】

はとにかく戻る。errcodeパラメータは下記の考えられる結果を含む。

【数 6 0】

CHAN\_SUCCESS  
CHAN\_CHANNEL\_NOT\_FOUND  
CHAN\_BUFFER\_UNAVAILABLE

【0 0 6 4】

【数 6 1】

【数 6 2】

CHAN\_SUCCESS  
CHAN\_CHANNEL\_NOT\_FOUND  
CHAN\_BUFFER\_CTRL\_ERROR

【0 0 6 5】

【数 6 3】

【0 0 6 6】

【数 6 5】

CHAN\_SUCCESS  
CHAN\_DEVID\_NOT\_FOUND  
CHAN\_DEVID\_NOT\_RESPONDING  
CHAN\_TASK\_NOT\_FOUND  
CHAN\_CHANNEL\_NOT\_FOUND

【0 0 6 7】 ストリーム

ストリームはフレームに押し入ることはできないが、タスクに対して連続的に流出入するデータのために使用される。ストリームはヘッドおよび流出入時にデータを追跡するテールポインターを付随するサーキュラーバッファ（FIFO）からなっている。各タスクは指定され

た入力および出力ストリームを有することができる。ホスト-DSPインターフェイスを横切るストリームデータフローは次のようである。

【数 67】

StrmPktFlag	Stream ID	Count	Data[...]
-------------	-----------	-------	-----------

ストリーム関数を下記に示す。

【数 68】

【0068】

#### Create\_Stream

```
BOOL Create_Stream(T_DeviceID DevID, T_Size FIFOsize, T_StrmID
*StreamID, U16 *errcode);
```

この関数はFIFOベース通信ストリームを生成する。それはサイズFIFOsizeのFIFOのコントロールを維持するストリームコントロール構造を生成する。生成されると、ストリームは空FIFOを割り当て、ストリームに流出入するデータフローを処理するようにヘッドおよびテールポインターを初期化する。StreamIDは新しいストリームのIDを返す。DevIDが呼出プロセッサのものでなければ、呼出プロセッサおよびDevIDプロセッサの両方にストリームコントロール構造が生成され、通信インターフェイスを横切って流

れるデータをコントロールする。errcodeパラメータは下記の考えられる結果を含む。

【数 69】

```
STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_ALLOCATION_ERROR
```

【0069】

【数 70】

#### Delete\_Channel

```
BOOL Delete_Stream(T_StrmID StreamID, U16 *errcode);
```

この関数はStreamIDにより指定される既存のストリームを削除する。errcodeパラメータは下記の考えられる結果を含む。

【0070】

【数 72】

【数 71】

```
STRM_SUCCESS
STRM_DEVID_NOT_FOUND
STRM_DEVID_NOT_RESPONDING
STRM_STREAM_NOT_FOUND
```

#### Get\_Stream\_Count

```
BOOL Get_Stream_Count(T_LocalStrmID StrmID, T_Count *count, U16
*errcode);
```

この関数は現在StrmIDにより指定されるストリームFIFO内にあるT\_DSP\_Wordsのカウントを要求する。countパラメータは戻った時に数を含んでいる。errcodeパラメータは下記の考えられる結果を含む。

【数 73】

```
STRM_SUCCESS
STRM_STREAM_NOT_FOUND
```

【0071】

【数 74】

#### Write\_Stream

```
BOOL Write_Stream(T_LocalStrmID Strm, T_DSP_Word *bufptr, T_Count
count, T_Count *countwritten, U16 *errcode);
```

この関数はStrmにより指定されたストリームへT\_DSP\_Wordsのcount数を書き込む。StrmがNULLであれば、タスクの出力ストリームが使用さ

れる。データはbufptrパラメータにより指示される。戻った時に、countwrittenは実際に書き込まれるT\_DSP\_Words数を含んでい

る。errcodeパラメータは下記の考えられる結果を含む。

【0072】

【数76】

【数75】

STRM\_SUCCESS  
STRM\_DEVID\_NOT\_FOUND  
STRM\_DEVID\_NOT\_RESPONDING  
STRM\_STREAM\_NOT\_FOUND  
STRM\_STREAM\_OVERFLOW

#### Read\_Stream

*BOOL Read\_Stream(T\_LocalStrmID Strm, T\_DSP\_Word \*bufptr, T\_Count maxcount, BOOL WaitFlag, T\_Count \*countread, U16 \*errcode);*

この関数はStrmにより指定されたストリームからデータを読み出す。StrmがNULLであれば、タスクの入力ストリームが使用される。データはbufptrにより指定されたバッファ内に格納される。maxcountまでTDSP\_Wordsがストリームから読み出される。countreadパラメータは読み出したデータの実際のカウントを含んでいる。errcodeパラメータは下記の考えられる結果を含む。

【数77】

STRM\_SUCCESS  
STRM\_DEVID\_NOT\_FOUND  
STRM\_DEVID\_NOT\_RESPONDING  
STRM\_STREAM\_NOT\_FOUND

【0073】

【数78】

#### Set\_Task\_Input\_Stream

*BOOL Set\_Task\_Input\_Stream(T\_Task \*TaskID, T\_StrmID StrmID, U16 \*errcode);*

この関数は指定されたストリームIDへタスクの入力ストリームを設定する。errcodeパラメータは下記の考えられる結果を含む。

【0074】

【数80】

【数79】

STRM\_SUCCESS  
STRM\_DEVID\_NOT\_FOUND  
STRM\_DEVID\_NOT\_RESPONDING  
STRM\_TASK\_NOT\_FOUND  
STRM\_STREAM\_NOT\_FOUND

#### Set\_Task\_Output\_Stream

*BOOL Set\_Task\_Output\_Stream(T\_Task \*TaskID, T\_StrmID StrmID, U16 \*errcode);*

この関数は指定されたストリームIDへタスクの出力ストリームを設定する。errcodeパラメータは下記の考えられる結果を含む。

【0075】

【表3】

【数81】

STRM\_SUCCESS  
STRM\_DEVID\_NOT\_FOUND  
STRM\_DEVID\_NOT\_RESPONDING  
STRM\_TASK\_NOT\_FOUND  
STRM\_STREAM\_NOT\_FOUND



表 8 データタイプ  
ここで使用するデータタイプを表 8 に示す。

シンボル	説明
S8	Signed 8-bit integer. (署名 8 ビット整数)
U8	Unsigned 8-bit integer. (非署名 8 ビット整数)
S16	Signed 16-bit integer. (署名 1 6 ビット整数)
U16	Unsigned 16-bit integer. (非署名 1 6 ビット整数)
S32	Signed 32-bit integer. (署名 3 2 ビット整数)
U32	Unsigned 32-bit integer. (非署名 3 2 ビット整数)
T_HostWord	A word on the Host processor. (ホストプロセッサ上の語)
T_DSP_Word	A word on the DSP processor. (DSP プロセッサ上の語)
BOOL	Boolean value(TRUE or FALSE). (ブール値(TRUE 又は FALSE))
T_HostPtr	Pointer on the Host processor. (ホストプロセッサ上のポインター)
T_DSP_Ptr	Pointer on the DSP processor. (DSP プロセッサ上のポインター)
T_DeviceID	Processor device ID. (プロセッサデバイス ID)
T_TaskID	A structure containing fields for a device ID and a processor local task ID. (デバイス ID 及びプロセッサローカルタスク ID 用フィールドを含む構造)
T_ChannelID	A structure containing fields for a device ID and a processor local channel ID. (デバイス ID 及びプロセッサローカルチャンネル ID 用フィールドを含む構造)
T_MsgID	Message ID. (メッセージ ID)
T_DSP_ID	An object ID on the DSP. (DSP 上のオブジェクト ID)
T_Count	Data type for a count. (カウン트의データタイプ)
T_Size	Data type for a size. (サイズのデータタイプ)
T_HostCallBack	Value used when tasks send message back to the Host. (タスクがホストへメッセージを送信する時の値)
T_ReplyRef	Message reply reference. (メッセージ返答参照)
T_LocalTaskID	Local task ID. (ローカルタスク ID)
T_LocalChanID	Local channel ID. (ローカルチャンネル ID)
T_DSP_Name	Name for DSP objects(RTOS dependent). (DSP オブジェクト名 (RTOS 依存))
T_CodeHdr	Code header structure for a DSP Library entry. (DSP ライブラリエントリ用コードヘッダー構造)
T_TaskCreate	Task creation structure. (タスク生成構造)

#### 【0076】システムメッセージ

これらの表はデバイス間を通過するメッセージを定義する(すなわち、ホストから DSP 16 へ)。メッセージをデバイスへ実際にルーティングするのに使用されるため、対応する関数呼出し内にパラメータとして存在するデバイス ID はメッセージには内蔵されない。同様に、

関数呼出しの上半部としてデバイス ID を含むタスク ID はメッセージ内にデバイス ID を含まず、DSP のロードタスク ID 部しか含んでいない。

#### 【0077】

【表 4】

表 4 DSP-API メッセージ

メッセージ	パラメータ送付	パラメータ返答	方向 Host ↔ DSP
GET MIPS	None	U32 mips	→
GET_MEM_AVAIL		T_Size progmem T_Size datamem	→
ALLOC_MEM	U16 mempage T_Size size	T_DSP_Word *memptr U16 errcode	→
FREE_MEM	U16 mempage T_DSP_Word *memptr	U16 errcode	→
PUT_BLOB	T_DSP_Ptr destaddr U16 mempage T_Size size T_DSP_Word BLOB[size]	U16 errcode	→
CREATE_TASK	T_TaskCreate tcs	T_TaskID TaskID U16 errcode	→
START_TASK	T_TaskID TaskID	U16 errcode	→
SUSPEND_TASK	T_TaskID TaskID	U16 errcode	→
RESUME_TASK	T_TaskID TaskID	U16 errcode	→
DELETE_TASK	T_TaskID TaskID	U16 errcode	→
CHANGE_PRIORITY	T_TaskID TaskID U16 newpriority	U16 oldpriority U16 errcode	→
GET_TASK_STATUSES	T_TaskID TaskID	U16 status U16 priority T_ChainID Input T_ChainID Output U16 errcode	→
GET_ID	T_DSP_Name Name	T_DSP_ID ID U16 errcode	→

【0078】

【表5】

表 5 インターフェイスレイヤ/チャンネルインターフェイスレイヤメッセージ

メッセージ	パラメータ送付	パラメータ返答	方向 Host ↔ DSP
CREATE_CHANNEL	T_Size framesize T_Count numframes	T_ChainID ChannelID U16 errcode	→
DELETE_CHANNEL	T_ChainID ChannelID	U16 errcode	→
CREATE_STREAM	T_Size FIFOsize	T_StreamID StreamID U16 errcode	→
DELETE_STREAM	T_StreamID StreamID	U16 errcode	→

【0079】

【表6】

表 6 デバッグメッセージ

メッセージ	パラメータ送付	パラメータ返答	方向 Host ↔ DSP
READ_MEM	U16 mempage T_DSP_Ptr addr T_Count count	T_DSP_Word mem[count] U16 errcode	→
WRITE_MEM	U16 mempage T_DSP_Ptr addr	U16 errcode	→
	T_Count count T_DSP_Word mem[count]		
READ_REG	U16 RegID	DSP_WORD regvalue U16 errcode	→
WRITE_REG	U16 RegID T_DSP_Word regvalue	U16 errcode	→
SET_BREAK	T_DSP_Ptr addr	U16 errcode	→
CLR_BREAK	T_DSP_Ptr addr	U16 errcode	→
BREAK_HIT	T_DSP_Ptr addr	U16 ACK	←

【0080】ダウンロードネイティブコード

図4-図6はネイティブコードをターゲットプロセッサ（すなわち、ホスト12もしくはDSP16）へ確実にかつ効率的にダウンロードする実施例を示す。コードをダウンロードするこの実施例は、例えば、インターネット、もしくは他のグローバルネットワーク、ローカルもしくはワイドエリアネットワーク、もしくはPCカードやスマートカード等の周辺装置からコードをダウンロードするのに使用できる。

【0081】図4にJAVA Bean90の実施例を示し、Bean90はネイティブコード92のラッパー

(wrapper)として作用する。BeanはさらにCode Type属性94a、Code Size属性94bおよびMISP要求属性94cとして表記されているいくつかの属性94を含んでいる。Bean90はLoad Codeアクション96a、Load Parametersアクション96bおよびExecute Parameter96cを含むいくつかのアクション96を有している。

【0082】動作において、Load Codeアクション96aは外部ネイティブコード（ターゲットプロセッサにとってネイティブ）をBean内へロードするの

に使用される。J A V A B e a n s はパーシステンス (p e r s i s t e n c e) を有するため、B e a n 9 0 はネイティブコード9 2 および属性9 4 を含むその内部状態を格納することができる。L o a d P a r a m e t e r s アクション9 6 b はネイティブコード9 2 から (例えば、前記したC O F F ファイルフォーマットを使用して) パラメータを検索し属性9 4 a - c として格納する。E x e c u t e アクション9 6 c はD S P 1 6 内にインストールされたタスクを実行する。

【0083】B e a n 9 0 を使用してコードをターゲットプロセッサへダウンロードする様子を図5に示す。この例では、ターゲットプロセッサはD S P 1 6 (もしくは、多数のD S P 1 6 の中の1つ) であるものとするが、それはネイティブコードをホストプロセッサ1 2 へダウンロードするのにも使用できる。さらに、所望のB e a n 9 0 はL A N サーバもしくはインターネットサーバ等のネットワークサーバ内に常駐しているものとするが、B e a n はスマートカード等のプラットフォーム1 0 と通信する任意のデバイス内に常駐することができる。ワイヤレスデータプラットフォーム1 0 については、ネットワークサーバ1 0 0 への接続はしばしばワイヤレスである。

【0084】図5において、プラットフォーム1 0 はネットワークサーバ1 0 0 に接続されている。図2に詳細に示すホストプロセッサ1 2 はJ A V A バーマシ4 4 を介して1つ以上のJ A V A アプレット4 0 を実行することができる。新しいコードをダウンロードするために、ホスト1 2 はネットワークサーバ1 0 0 からB e a n 9 0 を含むアプレットをロードするか、もしくはアプレットを含まないB e a n をサーバ1 0 0 からダウンロードすることができる。ラッパーB e a n 9 0 が検索されると、ネイティブコードのサイズ、コードタイプ (コードはどのプロセッサ用か) および必要なM I P S を質問することができる。所期のプロセッサがコード9 2 を実行するのに十分な資源を有する場合には、コード9 2 は所期のプロセッサ、図5に示すアーキテクチャ内のホストプロセッサ1 2 もしくはD S P 1 6、で実行するようにインストールすることができる。典型的には、ネイティブコード9 2 は非リンク、コンパイルされたコードである。したがって、D S P - A P I 5 0 のクロスリンカー8 2 はコードを利用可能なメモリ位置へリンクする。B e a n は2進ネイティブコード9 2 をダイナミッククロスリンカー8 2 へ通し、それがコードをインストールして実行する。

【0085】典型的に、ネイティブコードのダウンロードが生じるのは、その中にD S P 関数が所望されるアプレットをユーザが実行している場合である。最初に、アプレットは所望のコードがタスク6 0 としてD S P 内にインストールされている、またはD S P ライブラリ5 8 内で利用可能であるかをチェックする。もしそうならタ

スクはダウンロードなしで実行できる。

【0086】タスクがD S P 1 6 もしくはD S P ライブラリ5 8 内に格納されていなければ、オブジェクト (ここでは、“D S P L o a d e r” オブジェクトと言う) を生成してB e a n 9 0 をロードすることができる。D S P L o a d e r クラスがホスト1 2 上でローカルであれば、J A V A はB e a n もローカルに利用可能であるかをチェックする。最初の場合、ローカルに格納されたコードを有するB e a n があることがある。そうであれば、B e a n からのコードはD S P (もしくは、C o d e T y p e により指定されるいずれかのプロセッサ) へインストールされる。コードのないB e a n がローカルに格納される場合には、B e a n は適切なサーバからコードを検索することができる。

【0087】一方、D S P L o a d e r オブジェクトがローカルではなければ、J A V A はアプレットを書き込んだサーバからB e a n 9 0 をロードする。次に、B e a n からのコードは前記したようにインストールされる。

【0088】J A V A B e a n の使用に関連してネイティブコードのダウンロードが説明されるが、A c t i v e X アプレット等の他の言語内にコードを包み込んで達成することもできる。

【0089】J A V A B e a n (もしくは、他のアプレット) をネイティブコードのラッパーとして使用することは著しい利点がある。最初に、複数のプロセッサの中の1つへコードをローディングするのに単純な標準的方法でよい。B e a n が生成され、コードがB e a n へロードされて適切なプロセッサとリンクされる。B e a n 内にコードをラッピングすることなく、プロセスは数百のステップをとることができる。第2に、多数のネイティブコード片を1つのアプレットにより結合し、1つのアプレットを使用する多数の個別のルーチンから複雑なアプリケーションを発生してルーチンを所望により結合することができる。第3に、言語のセキュリティ機能を利用することができ、B e a n 9 0 内のJ A V A コードだけでなくネイティブコード9 2 も保護される。A c t i v e X 等の他の言語もセキュリティ機能を有する。

#### 【0090】セキュリティ

2つの最重要セキュリティ機能はデジタル署名および暗号化である。J A V A B e a n もしくはA c t i v e X アプレットはコードソースにより署名することができ、B e a n もしくはアプレットがダウンロードされると、署名は信頼できるソースのリストを有する受信アプリケーションにより照合される。B e a n もしくはアプレットに信頼できるソースの署名があれば、標準技術を使用して解読することができる。したがって、ネイティブコードは伝送中にB e a n もしくはアプレットのコードと共に暗号化され、コードの非認可修正が防止される。ネイティブコードは安全であり信頼できるソースから来る

ため、属性も正確であるものと信頼することができる。

【0091】図6はJ A V A B e a nを使用するプロセッサのネイティブコードダウンロードプロセスを説明するフロー図であり、ネイティブコードは同様な技術を使用して異なる言語のアプレットで包むことができることがお判りであろう。ステップ110において、暗号化されたデジタル署名B e a n 90はJ A V Aバーチャルマシンを動かすデバイスへダウンロードされる。ステップ112において、署名が照合される。信頼できるソースとして表記されたソースからのものでなければ、ステップ114において例外処理がイネーブル(e n a b l e)される。B e a nが信頼できるソースから来るものである場合、ユーザがそのソースで不自由でなければ、例外処理関数はユーザにB e a nを受け入れる機会を与えることができる。署名が無効であれば、例外処理はB e a n 90を削除してユーザへ適切なエラーメッセージを送ることができる。

```
package ti.dsp.loader;

import java.awt.*;
import java.io.*;
import java.net.*;

public class NativeBean extends Canvas implements Serializable
{
    public NativeBean() {

        setBackground(Color.white);

        funcData = new ByteArrayOutputStream();

        try {
            funcCodeBase = new URL("http://localhost");
        }
        catch (MalformedURLException e) {
```

【表8】

【0092】署名が有効で信頼できるソースから来るものであれば、ステップ116においてB e a nが解読される。このステップではJ A V AコードとB e a n内のネイティブコードの両方が解読される。ステップ118において、B e a n 90から属性が検索されステップ120において、アプレットは適切なプロセッサがコードを実行するのに十分な資源を有するかどうかを確認する。十分な資源を有しない場合には、例外処理ステップ114はネイティブコードのインストールを拒絶することができ、あるいは資源を解放するステップをとることができる。十分な資源がある場合には、ステップ122においてコードはクロスリンカーを使用してリンクされ所望のプロセッサにインストールされる。ステップ124において、ネイティブコードが実行される。

【0093】B e a n 90のサンプルJ A V Aスクリプトを下記に示す。

【表7】

```

    }
}

public Dimension getMinimumSize() {
    return new Dimension(50, 50);
}

public void loadCode() {
    URL baseUrl = null;

    try {
        baseUrl = new URL(funcCodeBase.toString() + "/" + myFunction);
    }
    catch (MalformedURLException e) {
    }

    DataInputStream source = null;
    int read;
    byte[] buffer;

    buffer = new byte[1024];
    try {
        source = new DataInputStream(baseUrl.openStream());
    }
    catch (IOException e) {
        System.out.println("IOException creating streams: " + e);
    }

    codeSize = 0;

    funcData.reset();

    try {
        while (true) {

            read = source.read(buffer);

            if (read == -1)
                break;

            funcData.write(buffer, 0, read);
        }
    }
    catch (IOException e) {
        System.out.println("IOException: " + e);
    }
}

```

【表 9】

```
        codeSize = funcData.size();
        System.out.println("Code size = " + codeSize);

        try {
            source.close();
        }
        catch (IOException e) {
            System.out.println("IOException closing: " + e);
        }
    }

    public synchronized String getFunctionName() {
        return myFunction;
    }

    public void setFunctionName(String function) {
        myFunction = function;
    }

    public synchronized String getCodeBase() {
        return funcCodeBase.toString();
    }

    public void setCodeBase(String newBase) {
        try {
            funcCodeBase = new URL(newBase);
        }
        catch (MalformedURLException e) {
        }
    }

    public void installCode() {
        FileOutputStream destination = null;
        File libFile = new File(myFunction);

        try {
            destination = new FileOutputStream(libFile);
        }
        catch (IOException e) {
            System.out.println("IOException creating streams: " + e);
        }

        if (destination != null) {
```

【表 1 0】

```

        try {
            funcData.writeTo(destination);
        }
        catch (IOException e) {
            System.out.println("IO Exception installing native code: " + e);
        }
    }
}

linkCode(funcData)

public void loadParameters() {
}

public void execute() {
}

public synchronized int getCodeSize() {

    return codeSize;
}

public synchronized int getCodeType() {

    return codeType;
}

public void setCodeType(int newType) {

    codeType = newType;
}

private int codeSize = 0;
private int codeType = 1;
private String myFunction = "";
private URL funcCodeBase = null;
private ByteArrayOutputStream funcData = null;
}

```

【0094】前記したスクリプトにおいて、NativeBean () ルーチンはネイティブコードを保持するBean90を生成する。loadCode () ルーチンはサーバからネイティブコードを得る。getFunctionalName () およびgetCodeBase () ルーチンは属性を検索する。installCode () ルーチンはクロスリンカーを呼び出してネイティブコードをDSPにリンクしリンクしたコードをロードする。loadParameters () ルーチンはネイティブコードを調べてその属性を確認するようBeanを命令する。getCodeSize () およびgetCodeType () ルーチンは属性を要求するアプレットへ転送する。

【0095】本発明の詳細説明はある代表的な実施例に向けられたが、当業者ならば別の実施例だけでなくこれらの実施例のさまざまな修正が考えられるであろう。請

求の範囲に入る修正および別の実施例は全て本発明に含まれるものとする。

【0096】以上の説明に関して更に以下の項を開示する。

(1) 移動電子装置であって、ネイティブコードを実行するコプロセッサと、ホストプロセッサシステムおよびプロセッサ独立コードに対応するネイティブコードを実行するように作動するホストプロセッサシステムであって、デジタル信号コプロセッサにより実施されるタスクをダイナミックに変化させるように作動するホストプロセッサシステムと、前記ホストプロセッサシステムと前記コプロセッサ間の通信回路と、を含む移動電子装置。

【0097】(2) 第1項記載の移動電子装置であって、前記コプロセッサはデジタル信号処理装置である、移動電子装置。

【0098】(3) 前記いずれか1項記載の移動電子装

置であって、前記プロセッサ独立コードは J A V A を含む、移動電子装置。

【0099】(4) 前記いずれか 1 項記載の移動電子装置であって、前記ホストプロセッサシステムは前記コプロセッサのネイティブコードを発生することができる、移動電子装置。

【0100】(5) 前記いずれか 1 項記載の移動電子装置であって、前記ホストプロセッサシステムはプロセッサ独立ソースコードをコンパイルすることにより前記コプロセッサのネイティブコードを発生することができる、移動電子装置。

【0101】(6) 前記いずれか 1 項記載の移動電子装置であって、前記ホストプロセッサシステムはソースコードの識別されたブロックをコンパイルする、移動電子装置。

【0102】(7) 前記いずれか 1 項記載の移動電子装置であって、前記ホストプロセッサシステムはコプロセッサで実行できるソースコードのブロックを識別してコードの前記ブロックをコンパイルする、移動電子装置。

【0103】(8) 前記いずれか 1 項記載の移動電子装置であって、さらに前記コプロセッサへダウンロードして実行することができるルーチンのライブラリを格納するメモリを含む、移動電子装置。

【0104】(9) 前記いずれか 1 項記載の移動電子装置であって、さらにハードウェア言語アクセラレータを含む、移動電子装置。

【0105】(10) 前記いずれか 1 項記載の移動電子装置であって、前記ハードウェアアクセラレータは J A V A アクセラレータを含む、移動電子装置。

【0106】(11) 第 1 項記載の移動電子装置であって、さらにネットワークからデータを受信するネットワークインターフェイス回路を含む、移動電子装置。

【0107】(12) 移動電子装置の制御方法であって、コプロセッサにおいてネイティブコードを実行するステップと、ホストプロセッサシステムにおいてネイティブコードおよびプロセッサ独立コードを実行するステップと、デジタル信号コプロセッサにより実施されるタスクを前記ホストプロセッサシステムによりダイナミックに変えるステップと、前記ホストプロセッサシステムと前記コプロセッサ間で通信を行うステップと、を含む方法。

【0108】(13) 第 1 2 項記載の方法であって、コプロセッサにおいてネイティブコードを実行する前記ステップは、デジタル信号処理装置においてネイティブコードを実行するステップを含む、方法。

【0109】(14) 第 1 2 項、第 1 3 項記載の方法であって、さらに前記一般的処理システムにおいて前記コプロセッサのネイティブコードを発生するステップを含む、方法。

【0110】(15) 第 1 4 項記載の方法であって、ネ

ィティブコードを発生する前記ステップはプロセッサ独立ソースコードをコンパILINGしてネイティブコードを発生するステップを含む、方法。

【0111】(16) 第 1 5 項記載の方法であって、さらに前記ソースコードのブロックを識別して前記コプロセッサで実行するようコンパイルするステップを含む、方法。

【0112】(17) 第 1 2 項—第 1 6 項記載の方法であって、さらに前記ホストプロセッサシステムから前記コプロセッサへダウンロードして実行するルーチンのライブラリを格納するステップを含む、方法。

【0113】(18) 移動電子装置であって、複数のコプロセッサと、ホストプロセッサシステムであって、ソースコードを実行し、前記 1 個以上のコプロセッサで実行されるソースコードの 1 つ以上のセクションを識別し、ソースコードの識別した各セクションについて、対応するコプロセッサを決定し、ソースコードの識別した各セクションについて、コードの前記識別したセクションを前記対応するコプロセッサに関連するネイティブコードへコンパイルして前記対応するコプロセッサへインストールする、ように作動する前記ホストプロセッサシステムと、前記ホストプロセッサシステムと前記コプロセッサ間の通信回路と、を含む、移動電子装置。

【0114】(19) 移動電子装置の制御方法であって、ホストプロセッサシステムでソースコードを実行するステップと、1 個以上のコプロセッサで実行されるソースコードの 1 つ以上のセクションを識別するステップと、ソースコードの識別した各セクションについて、対応するコプロセッサを決定するステップと、ソースコードの識別した各セクションについて、コードの前記識別したセクションを前記対応するコプロセッサに関連するネイティブコードへコンパILINGして前記ネイティブコードを前記対応するコプロセッサへインストールするステップと、前記ホストプロセッサシステムと前記コプロセッサ間の通信を行うステップと、を含む、方法。

【0115】(20) 複数のプロセッサ (12, 16) を含むワイヤレスデータプラットフォーム (10)。タスクが実施される時に情報を通信できるようにプロセッサ間に通信チャネルが開設される。1 個のプロセッサで実施されるダイナミッククロスコンパイラ (80) がコードを別のプロセッサのためのネイティブ処理コードへコンパイルする。ダイナミッククロスリンカー (82) が他のプロセッサのためにコンパイルされたコードをリンクする。ネイティブコードはそれを包む J A V A Bean (90) (もしくは他の言語タイプ) を使用してプラットフォームへダウンロードすることもできる。J A V A Bean はセキュリティのために暗号化しデジタル署名することができる。

【0116】関連出願の相互参照

本出願は同じ日付で出願され本開示の一部としてここに



組み入れられている、ウースリー等の米国特許出願第08/995,600号“Mobile Communication System with Cross Compiler and Cross Linker”（アットニードケット第26453号）、ビューワーの米国特許出願第08/995,597号“Method and Apparatus for Providing Downloadable Functionality to an Embedded Coprocessor”（アットニードケット第26440号）、およびビューワーの米国特許出願第08/995,603号“Method and Apparatus for Extending Security Model to Native Code”（アットニードケット第26439号）に関連している。

【図面の簡単な説明】

【図1】一般的なワイヤレスデータ処理に特に適したプラットフォームアーキテクチャのブロック図。

【図2】図1のプラットフォームの機能ブロック図。

【図3】ダイナミッククロスコンパILINGおよびダイナミッククロスリンキング機能の機能ブロック図。

【図4】デバイスへダウンロードするJ A V A Beanラッパー内に含まれたプロセッサで実行するネイティブコードの実施例。

【図5】遠隔サーバ上に配置されたJ A V A Beanからデバイス上のプロセッサへ含まれたネイティブコードを転送する操作を示す図。

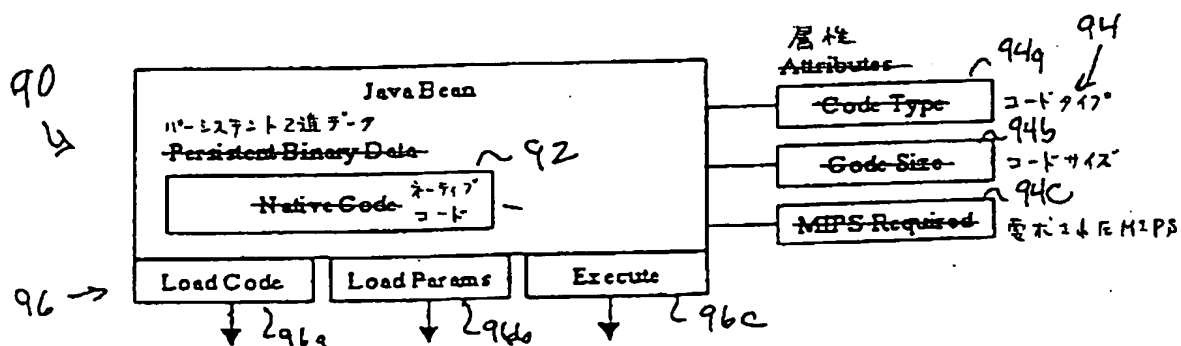
【図6】図5の操作に関連するセキュリティ機能を記述するフロー図。

【符号の説明】

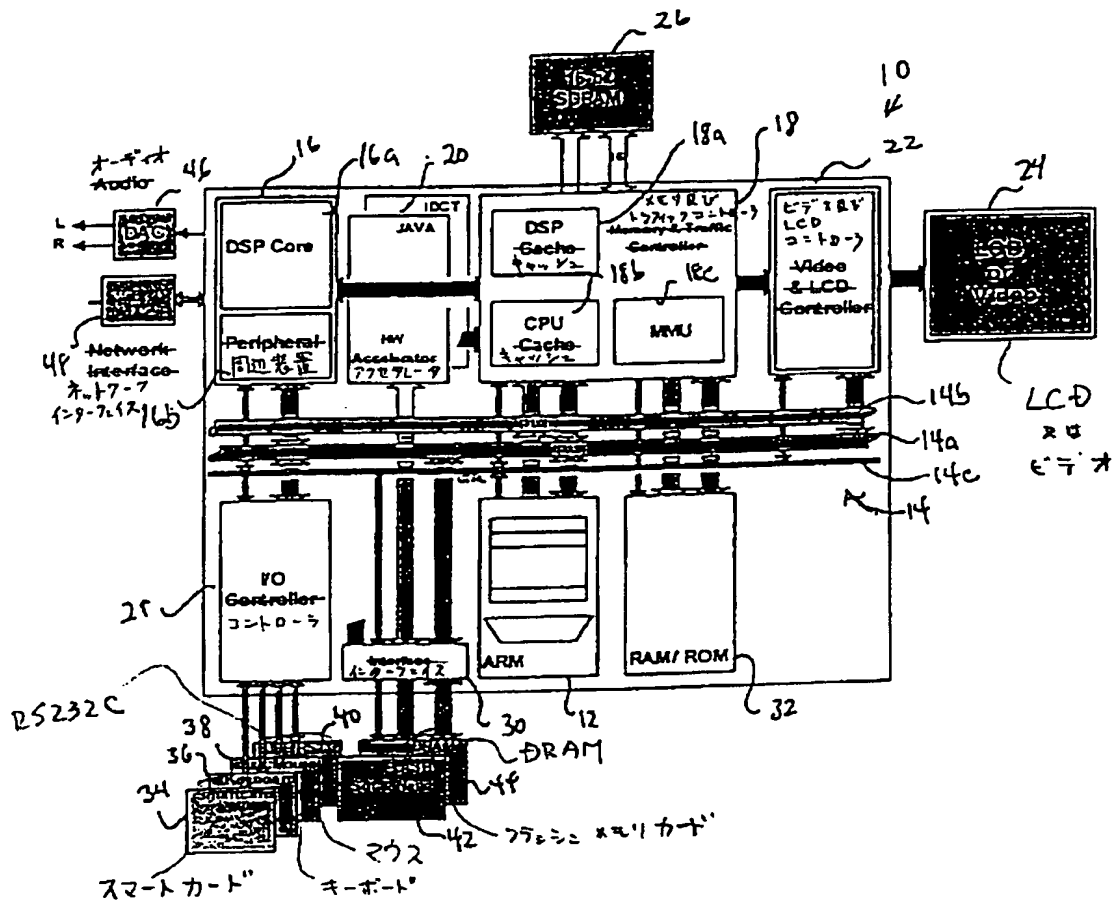
- 10 ワイヤレスデータプラットフォーム
- 12 汎用ホストプロセッサ
- 14 バス構造
- 16 DPS（デジタル信号処理装置）

- 18 トラフィックコントローラ
- 20 ハードウェアアクセラレータ回路
- 22 ビデオおよびLCDコントローラ
- 24 LCDもしくはビデオディスプレイ
- 26 主記憶装置
- 28 I/Oコントローラ
- 30 インターフェイス
- 32 RAM/ROM
- 34 スマートカード
- 36 キーボード
- 38 マウス
- 40 シリアルポート
- 42 フラッシュメモリカード
- 44 DRAMカード
- 46 DAC（デジタル/アナログコンバータ）
- 48 ネットワークインターフェイス
- 50 DSP API（アプリケーションプログラムインターフェイス）
- 52 ホストDSPインターフェイスレイヤ
- 54 DSPデバイスドライバ
- 56 ホストRTOS（リアルタイムオペレーティングシステム）
- 58 DSPライブラリ
- 60 タスク
- 62 ホスト-DSPインターフェイスレイヤ
- 64 DRTOS
- 66 ホストデバイスドライバ
- 70 ホスト-DSPインターフェイス
- 80 ダイナミッククロスコンパイラ
- 82 ダイナミッククロスリンカー
- 90 J A V A Bean
- 92 ネイティブコード
- 94 属性
- 96 アクション
- 100 ネットワークサーバ

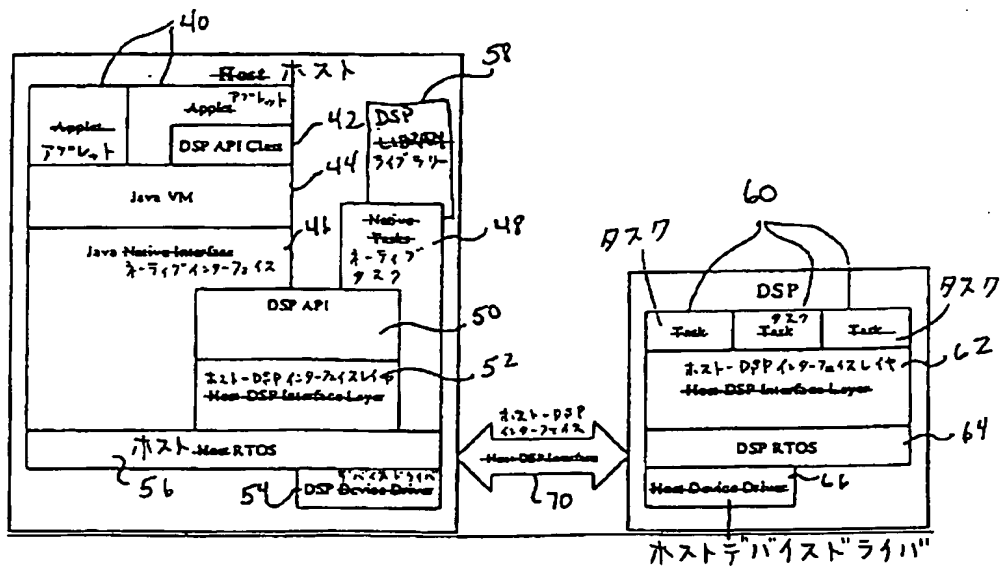
【図4】



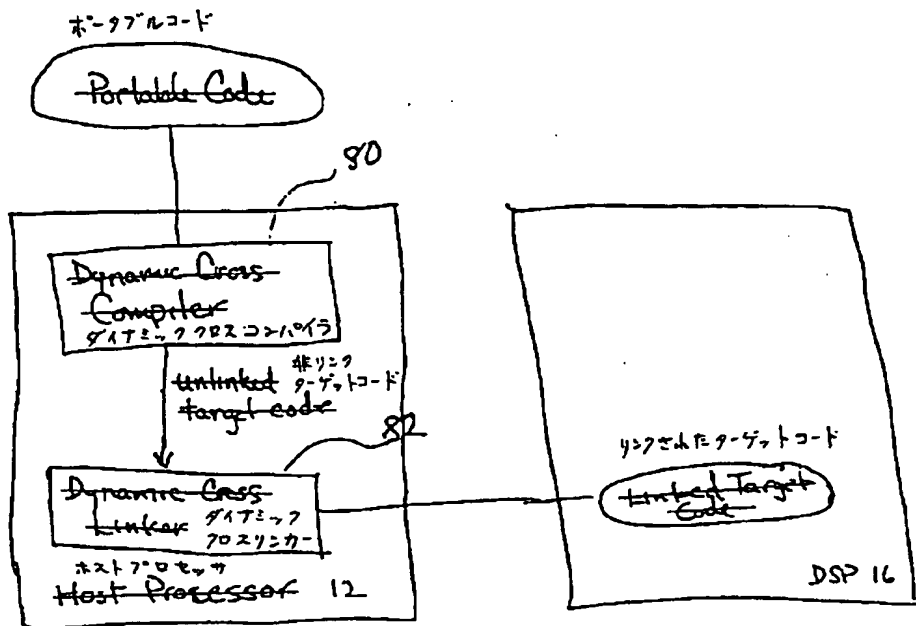
【図1】



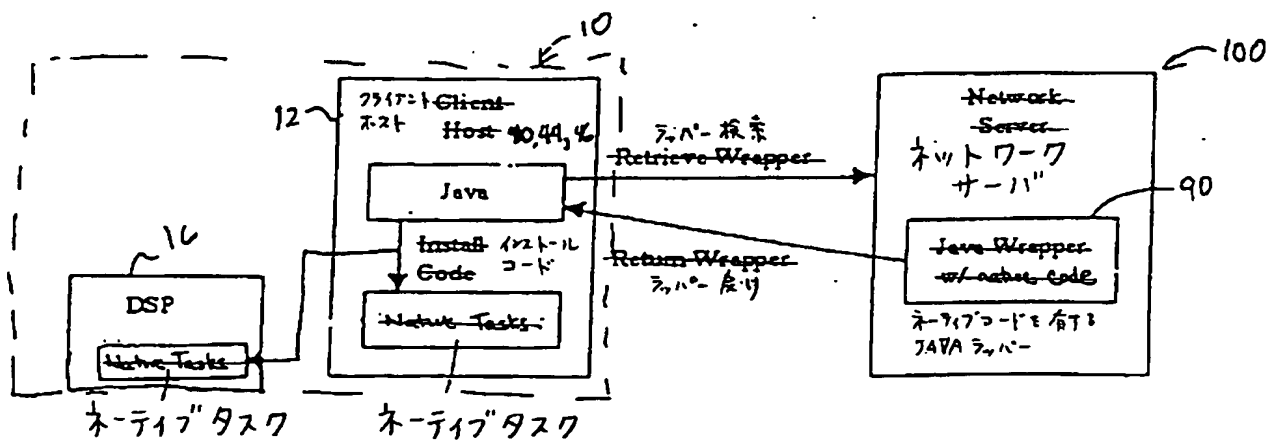
【図2】



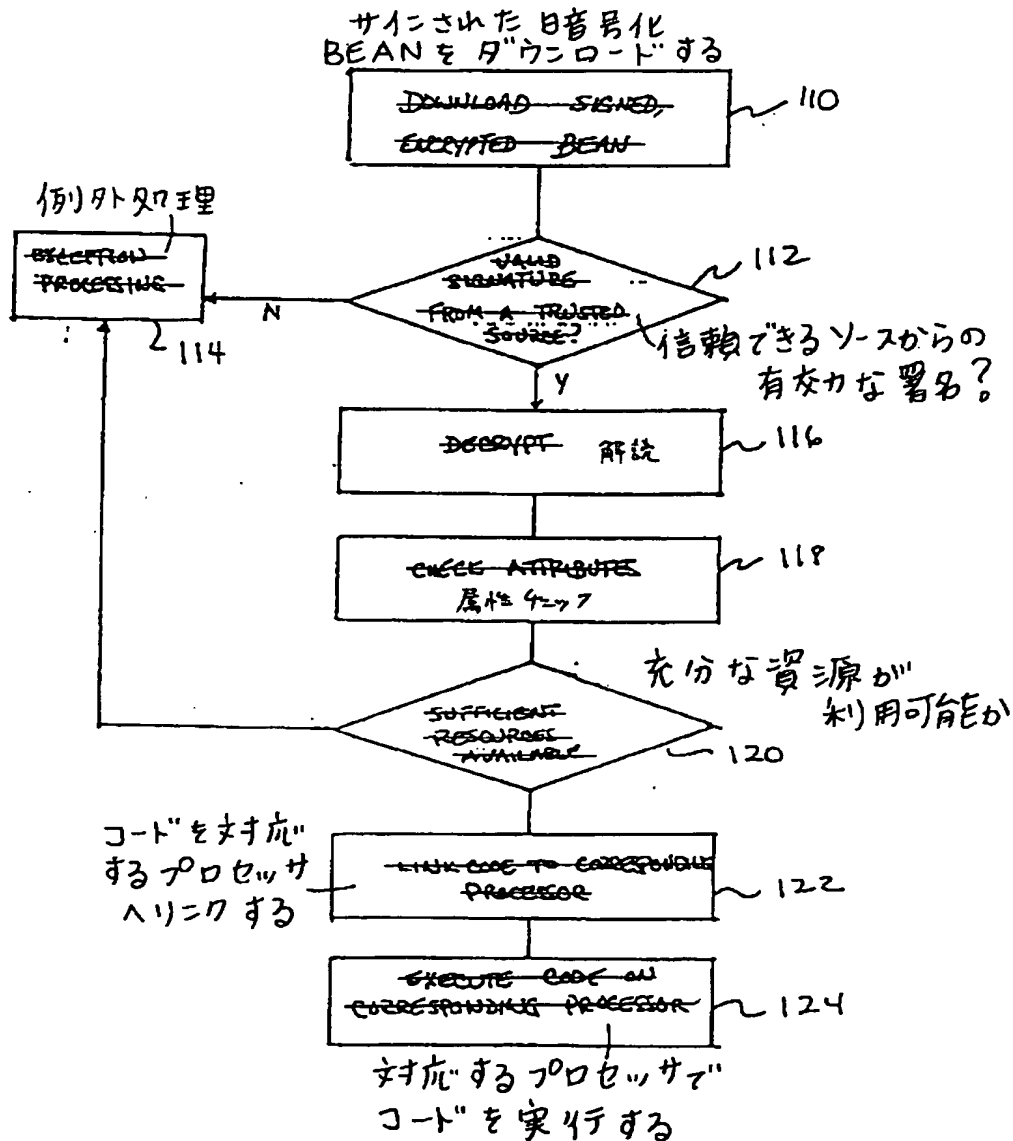
【図3】



【図5】



【図6】



フロントページの続き

(72)発明者 マシュー エイ. ウールゼイ  
アメリカ合衆国 テキサス州プラノ, ミド  
ルコーブ 608

(72)発明者 ジェラルド ショベル  
フランス国 アンティープ, シュマン ド  
ラ シュケット, レ ベルジェルド  
バル コンスタンス